Kapitel 2
Jenkins für Eilige

Wir tun, was wir können.
Aber …
Können wir auch, was wir tun?

Das Kapitel, das Sie gerade aufgeschlagen haben, war für mich das schwierigste: Wie beginnt man vor einem leeren Blatt Papier die eigenen Gedanken zu sortieren, um sich verständlich mitzuteilen? Am liebsten möchte ich mit all meinem Wissen und meinen ganzen Ideen auf einmal herausplatzten. Das geht aber nicht. Wenn man beabsichtigt, ein stabiles Haus zu bauen, muss zuerst ein Fundament gelegt werden.

Für mich ist stets der beste Weg, mich in ein neues Thema einzuarbeiten, mit einem Beispiel zu experimentieren. Deswegen beginnen wir zum Einstieg auch gleich mit dem ersten Build-Job.

2.1 Jenkins in 10 Minuten

Um mit Jenkins etwas Vernünftiges anstellen zu können, benötigen Sie vier Dinge:

- ein Softwareentwicklungsprojekt, am besten in Java,
- eine zugehörige, plattformunabhängige Build-Logik
  (Shell-Skripte und Batch-Dateien zählen nicht als Build-Logik!),
- ein Sourcecode-Repository (am besten in Git,
  aber auch Subversion ist machbar) und
- die Installation eines Jenkins-CI-Servers.

Wie Sie zu einer funktionsfähigen Jenkins-Installation kommen, erfahren Sie in Kapitel 7. Springen Sie dorthin, falls der letzte Punkt ein Problem darstellt.

Die ersten drei Punkte können Sie ganz einfach adressieren, indem Sie auf das offizielle Jenkins-Repository auf GitHub (*https://github.com/jenkinsci/jenkins*) zurückgreifen.

Ganz recht! Wir werden in diesem Einführungskapitel mit Jenkins den aktuellen Codestand des Jenkins-Servers bauen. Natürlich können Sie auch jedes beliebige frei

verfügbare Software-Repository auf GitHub nehmen, solange Sie dort funktionierenden Code vorfinden. Oder greifen Sie auf Ihr eigenes Projekt zurück.

Lost in Translation

Um Jenkins zum Sprachwechsel zu bewegen, müssen Sie die Sprache des von Ihnen verwendeten Webbrowsers umstellen. Denn Jenkins kommt multilingual daher und richtet sich nach den Spracheinstellungen Ihres Webbrowsers.

Auch wenn es komfortabel ist, den Jenkins-CI-Server mit dem deutschen Sprachpaket zu verwenden, empfehle ich Ihnen, sich möglichst für eine englischsprachige Installation zu entscheiden.

Das wichtigste Argument dafür ist die Suche nach Lösungen, wenn Probleme auftreten. Wenn Sie dann die englischen Bezeichnungen als Suchbegriffe verwenden, ist die Wahrscheinlichkeit weitaus höher, in den einschlägigen Entwicklerforen Hilfe zu finden. Leider sind die Antworten dann auch oft nicht auf Deutsch.

Auch bei der Verwendung diverser Plug-ins kann es vorkommen, dass für sie keine Übersetzung verfügbar ist. Dann haben Sie in Ihrer Installation ein hübsches Gewirr aus Deutsch und Englisch.

Die Screenshots in diesem Buch habe ich aus den genannten Gründen bis auf wenige Ausnahmen in der englischen Sprachversion angefertigt.

Gehen wir einmal davon aus, dass Ihr Jenkins bereits so weit vorbereitet ist, dass Sie ein einfaches Java-Projekt mit einer Maven-Build-Logik bauen können. Sie sind bereits erfolgreich eingeloggt und haben wie in Abbildung 2.1 den Startbildschirm vor sich.

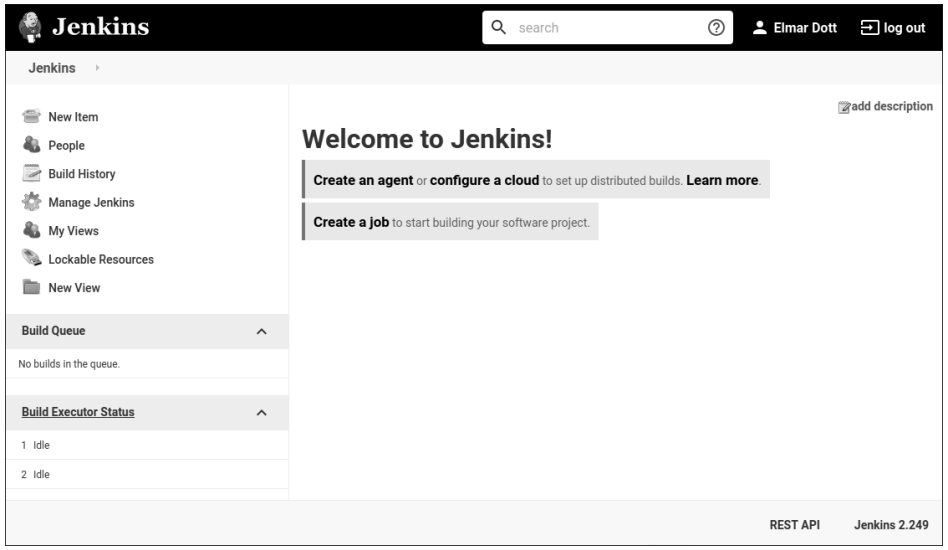

Abbildung 2.1 Der Startbildschirm nach erfolgreichem Login

Der erste Schritt ist, über den Menüpunkt NEW ITEM einen neuen Jenkins-Job vom Typ *Maven* zu erstellen. Abhängig von den installierten Plug-ins bietet sich Ihnen eine angepasste Auswahl der neu zu erzeugenden Job-Typen an (siehe Abbildung 2.2). Fehlt Ihnen beispielsweise der Punkt MAVEN PROJECT, müssen Sie das entsprechende Plug-in installieren. Die Installation des Jenkins-Servers und der einzelnen Plug-ins wird in Kapitel 7 beschrieben.

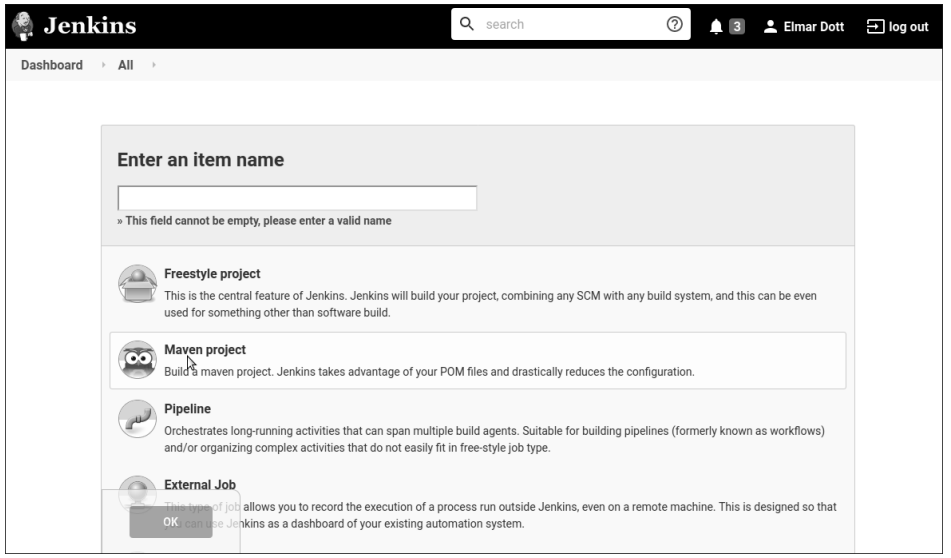

Abbildung 2.2 Auswahl der Jenkins-Job-Typen

Geben Sie Ihrem neuen Build-Job einen verständlichen Namen, und speichern Sie ihn über OK ab. In der Konfiguration, die sich dann öffnet, tragen Sie das GitHub-Repository ein, in unserem Fall also *https://github.com/jenkinsci/jenkins*.

Ein Benutzer für den Checkout der Dateien von GitHub wird nicht benötigt, da der anonyme Zugriff auf das Repository erlaubt ist. In Abbildung 2.3 ist ein Beispiel für die Konfiguration zu einem SCM-Checkout für GitHub dargestellt. Das bedeutet, dass wir die Funktion zum Sourcecode-Management von Jenkins nutzen, um den entsprechenden Code von der Plattform GitHub zu laden.

Dabei handelt es sich um einen lesenden Zugriff. Wenn Sie einen eigenen GitHub-Account besitzen, können Sie auch jedes beliebige Repository in Ihren GitHub-Bereich forken, d. h. abspalten. Damit haben Sie dann die Möglichkeit, in das abgespaltete Repository Codeänderungen zu übertragen. Das ist ein schreibender Zugriff, für den Sie Ihre Login-Informationen (*Credentials*) in Jenkins hinterlegen müssen. Wie Sie mit Credentials umgehen, erfahren Sie in Abschnitt 7.5.2.

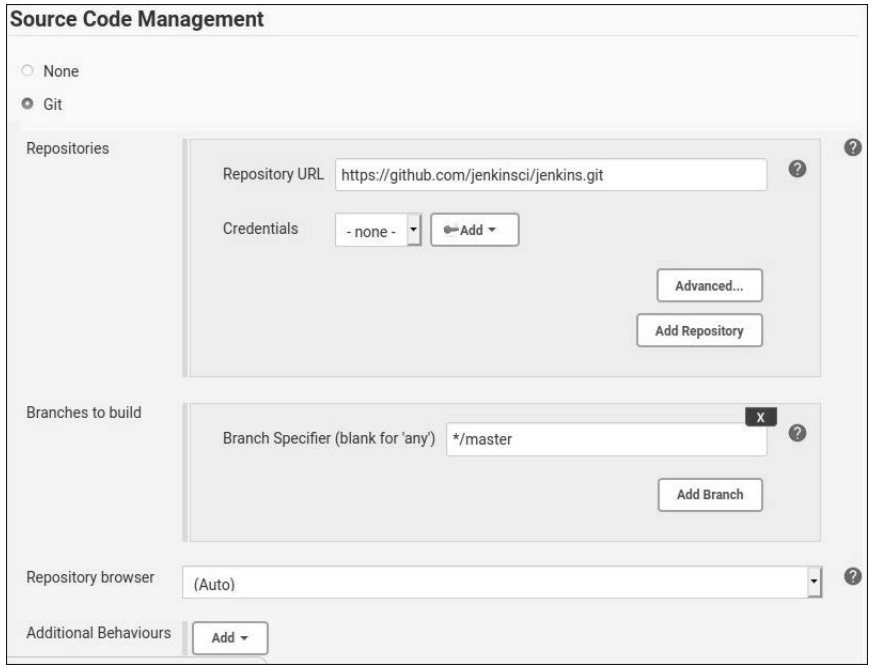

Abbildung 2.3 Job-Konfiguration für den GitHub-Checkout

Der nächste Punkt betrifft eine weitere sehr wichtige Einstellung, nämlich die Angabe der Build-Logik und wie diese gestartet wird.

Da es sich bei Jenkins um ein Maven-Projekt handelt, liegt die Build-Datei im Wurzelverzeichnis des Projekts und hat die Bezeichnung *pom.xml*.

Der Aufruf dieser POM-Datei erfolgt über das Kommando `verify` in der Zeile GOALS AND OPTIONS (siehe Abbildung 2.4). Um stets einen reproduzierbaren Build zu haben und unvorhergesehenes Verhalten zu verhindern, das durch alte Fragmente wie unvollständige Checkouts hervorgerufen werden kann, sollten Sie den Punkt DELETE WORKSPACE BEFORE BUILD STARTS auswählen – so sorgen Sie dafür, dass der Workspace des Projekts immer sauber ist. Aus Erfahrung bevorzuge ich bei sehr langlebigen Jenkins-Jobs, an denen ich lange arbeite, den Start mit einem frischen Projektverzeichnis, also dass das Projektverzeichnis vor der Ausführung der einzelnen Job-Schritte geleert wird. Sie können alternativ auch nach Beendigung des Jobs aufräumen und den Arbeitsbereich (Workspace) löschen. Das erschwert aber im Fehlerfall die Spurensuche nach Fehlern und Problemen: In einem leeren Verzeichnis können Sie sich nun mal nicht vergewissern, ob erwartete Kopieraktionen erfolgreich durchgeführt wurden.

Die Konfiguration speichern Sie direkt mit SAVE ab und navigieren so automatisch in den Build-Job hinein.

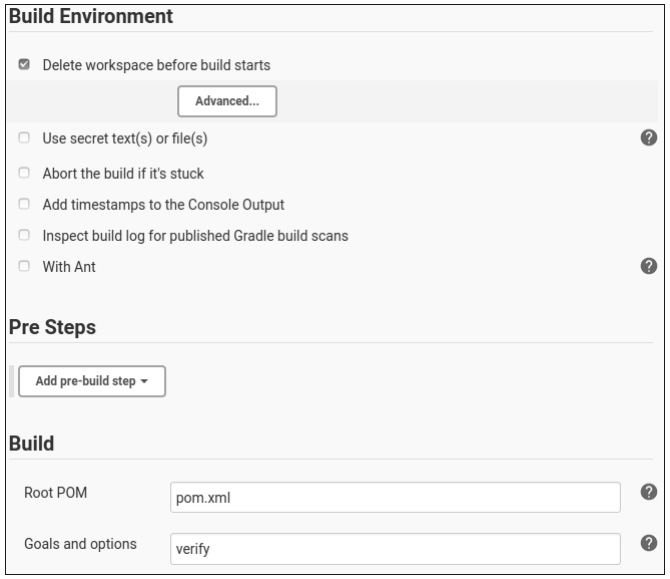

Abbildung 2.4 Job-Konfiguration für die Build-Umgebung

Von dort können Sie über den Punkt BUILD NOW den frisch angelegten Job starten. Wenn Sie alles korrekt eingestellt haben, sollte Ihr Build losgehen.

Nun benötigen Sie etwas Geduld. Je nachdem, wie leistungsfähig Ihr Computer ist, erhalten Sie früher oder später einen erfolgreich durchgelaufenen Build, wie er in Abbildung 2.5 zu sehen ist.

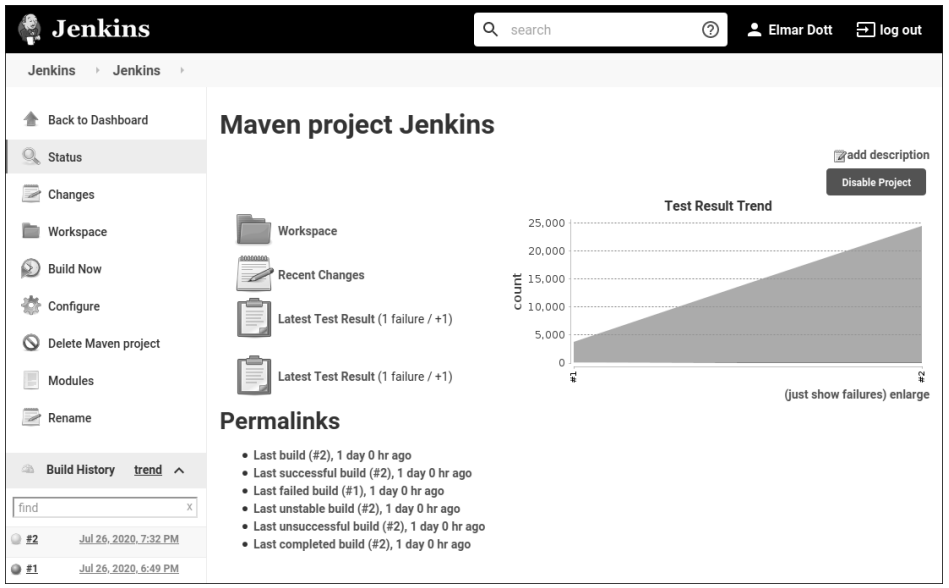

Abbildung 2.5 Übersichtsseite eines Build-Jobs

Neben den Optionen, einen Build zu starten und das Arbeitsverzeichnis zu inspizieren, wird Ihnen auch links unten eine BUILD HISTORY angezeigt. Dort erfahren Sie, dass zwei Jobs ausgeführt worden sind.

Die rote Kugel vor der Build-Nummer signalisiert einen Fehlschlag, den wir ein wenig später untersuchen werden. Widmen wir unsere Aufmerksamkeit vorerst dem zweiten Build-Job, der vollständig durchgelaufen ist. Mit einem Klick auf die Build-Nummer #2 gelangen Sie in die Statusübersicht des zweiten Builds, die Sie in Abbildung 2.6 sehen.

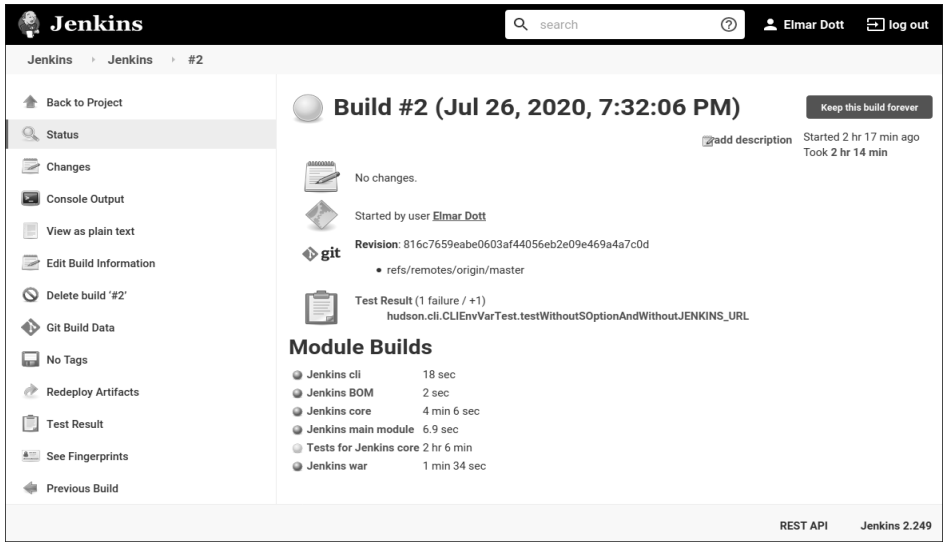

Abbildung 2.6 Übersicht eines fertig durchgelaufenen Build-Jobs

In der Übersicht erkennen wir oben rechts die gesamte Dauer des Builds von nahezu 140 Minuten – um genau zu sein: 2 Stunden und 14 Minuten. Wir erfahren auch gleich, dass der Hauptanteil der Zeit für die Ausführung der Testfälle benötigt wurde. Das Erzeugen der Datei *jenkins.war* benötigte nur 1:34 Minuten.

Die Farben der Modulübersicht verraten uns etwas mehr. Erfolgreich durchgelaufene Jobs bekommen den Farbstatus Blau. Als Europäer würde man eher eine grüne Einfärbung erwarten, da der Initiator des Jenkins-Servers aber Japaner ist und in anderen Ländern eben auch andere Sitten herrschen, ist die Wahl auf Blau gefallen.

Die gelbe Farbe vor den Testfällen, die auch für den gelben Gesamtstatus des Builds verantwortlich ist, signalisiert uns, dass nicht alle Testfälle erfolgreich durchlaufen wurden. Das deckt sich auch mit der Ausgabe in der Anzeige zu den Testergebnissen, die auf einen Fehler hinweist. So etwas kommt bei in der Entwicklung befindlichen Projekten häufiger vor und signalisiert, dass noch einiges an Arbeit zu verrichten ist,

bis ein Release entsteht. Die fertige *jenkins.war* finden Sie, wenn Sie in das Modul JENKINS WAR navigieren. Dort werden alle herunterladbaren Artefakte aufgelistet.

Schauen wir nun einmal, was den Fehler in Build #1 verursacht hat. Nach einem schnellen Wechsel in die Statusübersicht des Builds erhalten Sie weitere Informationen im Punkt CONSOLE OUTPUT. Dort gelangen Sie auch zur Logausgabe, die Sie in Abbildung 2.7 sehen.

```
[INFO] --- maven-enforcer-plugin:3.0.0-M3:enforce (enforce-versions) @ jenkins-war ---
[WARNING] Rule 1: org.apache.maven.plugins.enforcer.RequireJavaVersion failed with message:
Detected JDK Version: 1.8.0 is not in the allowed range [1.8.0-101,].
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for Jenkins main module 2.250-SNAPSHOT:
[INFO]
[INFO] Jenkins main module ............................... SUCCESS [01:27 min]
[INFO] Jenkins BOM ....................................... SUCCESS [  6.812 s]
[INFO] Jenkins cli ....................................... SUCCESS [ 50.292 s]
[INFO] Jenkins core ...................................... SUCCESS [06:03 min]
[INFO] Jenkins war ....................................... FAILURE [ 16.420 s]
[INFO] Tests for Jenkins core ............................ SKIPPED
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  11:35 min
[INFO] Finished at: 2020-07-26T19:02:19+02:00
```

Abbildung 2.7 Fehlermeldung nach dem ersten Durchlauf

Ziemlich gegen Ende des Logfiles finden Sie in aller Regel die Auflösung des Fehlers. In diesem Fall hat das Enforcer-Plug-in zugeschlagen, das sicherstellt, dass die Anwendung auch mit der richtiges Java-Version gebaut wird. In meinem Fall genügte das OpenJDK 8 den Anforderungen nicht.

Den Fehler habe ich behoben, indem ich in der Hauptkonfiguration von Jenkins als Default-JDK das OpenJDK 11 eingestellt habe. Eine Anleitung, die Ihnen zeigt, wie Sie die globale Konfiguration des JDK für Java-basierende Build-Umgebungen ändern, finden Sie in Abschnitt 7.3.

Die tatsächliche Magie des Bauens von Software findet sich in der Build-Logik – und die ist eine Wissenschaft für sich und muss zwangsläufig sehr individuell auf Ihr Projekt abgestimmt werden. Eine Übersicht über die wichtigsten Begriffe und Tools finden Sie in Kapitel 8.

Die bisherigen Einstellungen des angelegten Build-Jobs verlangen immer noch eine menschliche Interaktion für das Starten. Um tatsächlich kontinuierlich zu bauen, muss der Build automatisch getriggert werden. Das kann entweder regelmäßig geschehen oder durch eine bestimmte Aktion ausgelöst werden.

Der passende Schalter dazu findet sich in der Konfiguration des Jobs unter dem Punkt BUILD TRIGGER. Der Punkt POLL SCM ist eine der am häufigsten verwendeten

Optionen. Mit dem Eintrag @HOURLY veranlassen Sie Jenkins, stündlich in dem Git-Repository nach Änderungen zu schauen. Nur wenn es eine Änderung gibt, wird der Build auch tatsächlich gestartet.

Im Umgang mit Build-Jobs müssen Sie beachten, dass jeder Job sein eigenes *Arbeitsverzeichnis* hat. Diese Arbeitsverzeichnisse sind gut voneinander abgeschirmt, sodass man nicht ohne Weiteres hin und her wechseln kann. Eine Möglichkeit besteht darin, fertig gebaute Artefakte aus einem anderen Build-Job zu importieren – dazu müssen Sie aber ein Plug-in installieren. Und es besteht auch die Einschränkung, dass diese Option bei vielen Artefakten nicht sehr komfortabel zu nutzen ist. Diesen Details wenden wir uns in Kapitel 9 zu.

Hardware für erste Tests

Sicher fragen Sie sich, welche Hardware für eine Jenkins-Installation gut geeignet ist. Die mehr als zwei Stunden, die unser kleines Beispiel benötigt, sprechen ja schon eine sehr deutliche Sprache und machen klar, dass das Bauen von Software Ihr System sehr beanspruchen wird.

Für erste Gehversuche und die meisten der hier gezeigten Beispiele können Sie getrost auf eine lokale Installation auf Ihrem Rechner zurückgreifen.

Für den tatsächlichen produktiven Einsatz in Unternehmen ist diese Frage nicht so leicht zu beantworten. Aufwendige Jenkins-Jobs und umfangreiche Software-Projekte können schnell extrem ressourcenhungrig werden. Da ich im Einleitungskapitel nicht gleich ein ganzes Buch vorwegnehmen kann, verweise ich Sie an dieser Stelle auf Kapitel 7, in dem diese Punkte ein wenig detaillierter besprochen werden. Der Aufbau einer Jenkins-Installation, die produktiv genutzt werden soll, bedarf jedoch einer sehr individuellen Planung.

Kapitel 4
Software testen – aber wie?

Mit dem Testen können Fehler aufgedeckt werden, die Fehlerfreiheit kann jedoch nicht bewiesen werden.

Dass man Software testen muss, ist allen Beteiligten klar. Und auch die Detailfrage, welche Test-Frameworks eingesetzt werden, sollte uns zunächst keine Probleme bereiten. Vielmehr bereitet das *Wie* die meisten Unklarheiten. Grade im Umgang mit Werkzeugen wie Jenkins gehören wirkungsvolle Testautomatisierungen zu den essenziellen Konzepten, weswegen wir hier einen genaueren Blick auf das Thema werfen.

Das Feld der Softwaretests ist so umfangreich, dass damit mühelos ganze Bücher gefüllt werden. Die Möglichkeiten sind vielfältig und reichen von einfachen Unit-Tests über Akzeptanztests hin zu Integrationstests, die wir in diesem Kapitel besprechen.

- *Unit-Tests* oder *Komponententests* stellen die kleinste Testeinheit dar. Sie werden in Abschnitt 4.1 beschrieben.
- Was *Akzeptanztests* sind und wie diese automatisiert werden können, erfahren Sie in Abschnitt 4.2.
- Der Unterschied zwischen *Integrationstests* und Unit-Tests ist Thema des Abschnitt 4.3.

Bei diesen drei Testtypen handelt es sich also vornehmlich um die Bereiche des Testens, mit denen Entwicklerteams während ihrer täglichen Arbeit konfrontiert werden. Im Folgenden betrachten wir auch, wer für welche Tests zuständig ist und welches Paradigma dem zugrunde liegt. Weitere Testarten, wie Penetrationstests und Sicherheitschecks, werden uns allerdings nicht beschäftigen, da solche Audits im Regelfall von externen Spezialisten durchgeführt werden und zusätzlicher Vorbereitungen bedürfen.

Softwaretests verfolgen zwei Ziele: Die Validierung und die Verifikation.

- *Validierung* – Erstellen wir das richtige Produkt?
- *Verifikation* – Erstellen wir das Produkt richtig?

Das wichtigste Bedürfnis sämtlicher Prozeduren ist, nachzuweisen, dass sich der Prüfling (oft auch als *System Under Test*, SUT, bezeichnet) entsprechend der festge-

setzten Vorgaben verhält. Schließlich kann man es sich als Hersteller eines Onlineshop-Systems nicht leisten, dass die Applikation eine falsche Mehrwertsteuer errechnet. Ein solcher Fehler in produktiven Systemen kann zu erheblichen wirtschaftlichen Schäden für den Betreiber führen. Mit Sicherheit wird er deshalb den Hersteller in Regresspflicht nehmen. Um eine solche Situation zu vermeiden, wird der Hersteller durch Testen sicherstellen, dass die entwickelten Komponenten korrekt arbeiten. Es handelt sich also um eine wichtige Qualitätskontrolle.

In vielen Unternehmen werden diese Tests noch manuell durchgeführt. Das gestaltet sich dann meist so, dass nach der Bereitstellung eines Releases die Anwendung auf ein Testsystem installiert wird, und eine eigene Abteilung, die unabhängig von der Entwicklung ist, »klickt« sich durch das Programm. Um eine nachvollziehbare Abfolge zu ermöglichen, werden Testpläne formuliert, in denen aufgeschrieben wird, welche Aktionen welches Ergebnis bewirken sollen.

Dass ein solches Vorgehen einerseits teuer und ressourcenhungrig ist, ist leicht einzusehen. Viel schwerwiegender ist die Gefahr, dass sich durch Unaufmerksamkeit Fehler einschleichen. Schließlich neigen Menschen im Gegensatz zu Computern dazu, zu ermüden. Dieser Effekt wird durch monotone Arbeiten noch beschleunigt. Ein automatisiertes Vorgehen sorgt für mehr Stabilität und dient als lückenlose Dokumentation, dass ein ausgeliefertes Produkt tatsächlich die zugesicherten Eigenschaften aufweist.

Mit diesem Verfahren lässt sich allerdings nicht voraussagen, wie sich das System in Situationen verhält, die nicht durch Tests berücksichtigt wurden. Daher ist es wichtig, möglichst aussagekräftige Tests zu formulieren und mit diesen einen großen Bereich der Anwendung abzudecken.

Beim Testen von Systemen lassen sich grundsätzlich zwei Herangehensweisen unterscheiden:

- Ein *Black-Box-Test* beschreibt eine Situation, in der lediglich bekannt ist, wie eine Funktionalität sich verhält. Der Prüfer hat keine Kenntnis darüber, wie die Funktionalität umgesetzt wurde. Die Strategie bei dieser Testkategorie ist es, sämtliche zulässigen sowie unzulässigen Eingaben durchzuprobieren, um das Verhalten des SUT zu untersuchen.
- Ein *White-Box-Test* zeichnet sich dadurch aus, dass der Prüfer detaillierte Kenntnisse über die innere Struktur des Prüflings hat. In dieser Variante werden sämtliche Möglichkeiten, die das SUT durchlaufen kann, überprüft, wie es bei Unit-Tests üblich ist.

Es existieren auch Mischformen des Testens, die sich beide Ausprägungen zunutze machen. In diesem Fall spricht man vom *Grey-Box-Test*.

Unabhängig davon, welche Art von Tests man schreibt, hat sich ein einheitliches schematisches Vorgehen herausgebildet:

1. Bevor ein Test ausgeführt werden kann, müssen sämtliche Vorbedingungen (*Pre-Conditions*) erfüllt sein. Dabei handelt es sich um die korrekte Initialisierung von Variablen, Verbindungen zu Backend-Systemen wie Datenbanken und Ähnlichem.
2. Nach dem vollständigen *Arrange* in Schritt 1 folgt mit dem *Act* die Testausführung. Sie wird in wissenschaftlichen Aufsätzen hin und wieder als *Invariante* bezeichnet.
3. Zum Schluss erfolgt im dritten und abschließenden Schritt die Überprüfung (*Assume*) des Ergebnisses mit einem festgelegten Erwartungswert. Wurde die *Post-Condition* erfüllt und stimmt der Erwartungswert mit dem Ergebnis überein, gilt der Test als bestanden.

Kaputtrepariert

Um zu verhindern, dass Änderungen bereits fertiggestellte Funktionen negativ beeinflussen, bedient man sich sogenannter *Regressionstests*. Sie weisen also nach, dass immer noch alles funktioniert. Sämtliche in diesem Kapitel vorgestellten Methoden gehören laut dieser Definition ebenfalls in die Kategorie der Regressionstests, auch wenn diese nur manuell durchgeführt werden. Gründe für den Einsatz von Regressionstests sind:

- Änderung der Anforderung, die zu einer existierenden Funktionalität geführt haben
- Hinzufügen neuer Features
- Fehlerkorrekturen und Performance-Optimierungen
- Technologiewechsel oder Updates
- Code-Optimierungen

Man unterscheidet zwischen Regressionstests und Re-Tests. Zum *Re-Testen* kommt es dann, wenn in einer Funktion ein Fehler lokalisiert wurde und dieser behoben worden ist. Das Re-Testen stellt dann sicher, dass die Korrektur den Fehler tatsächlich beseitigt hat.

Es gibt bereits eine Vielzahl an einschlägiger Literatur über testgetriebene Softwareentwicklung (*Test-Driven Development*, TDD) und wie TDD zu deuten ist. Leider sind auch die Experten nicht immer einer Meinung. Viele Diskussionen sind oft sehr erregt und gleichen den »Religionskriegen«, die man von Themen wie »Was ist die beste Programmiersprache?« oder »Was ist die beste IDE?« her kennt. Ein Beispiel dazu ist die Testpyramide, die von Mike Cohen in seinem Buch »Succeeding with Agile« vorgestellt wird.

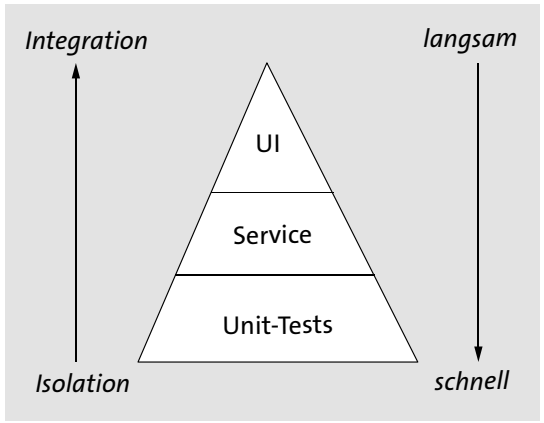

Abbildung 4.1 Testpyramide nach Mike Cohen

Die in der Pyramide beschriebenen drei Schichten (siehe Abbildung 4.1) finden sich in unserer Klassifizierung ebenfalls wieder. Ordnen wir daher die Schichten der Testpyramide unserer allgemein formulierten Struktur einmal zu:

1. **Komponententests** bilden die höchste Isolationsstufe und sind folglich am schnellsten ausführbar. Die englische Bezeichnung *Unit-Tests* ist den meisten etwas geläufiger.
2. **Akzeptanztests** bilden oft eine komplexe Prozedur ab und haben daher eine mittlere Ausführungsgeschwindigkeit. Wie in der Pyramide bereits angedeutet, handelt es sich hier um einfache oder zusammengesetzte *Services*. Die Login-Funktion stellt zum Beispiel eine solche Prozedur dar und kann auch als REST-Service implementiert sein.
3. **Integrationstests** bilden die Spitze der Pyramide und erfordern als minimale Bedingung, dass die Anwendung so weit erstellt wurde, dass sie zumindest ausgeführt werden kann. Hat man eine Tool-Suite wie Office-Paket, wäre ein vollständiger Integrationstest die Erstellung sämtlicher Komponenten wie Textverarbeitung und Tabellenkalkulation. Das alleinige Erstellen der Präsentationsanwendung wiederum ist nur eine Teilanwendung. Tests für die Benutzeroberfläche (UI) verlangen in aller Regel eine lauffähige Anwendung, weswegen diese bei den Integrationstests mit angesiedelt sind.

In dieser Reihenfolge werden wir uns nun den technischen Details widmen. Sämtliche Beispiele sind als Java- bzw. Maven-Projekt verfasst und können in den bekannten IDEs geöffnet und nachvollzogen werden. Die Beispiele sind einfach gehalten und lassen sich mit entsprechenden Anpassungen auch an andere Programmiersprachen und Werkzeuge adaptieren. Das zugehörige Repository finden Sie frei zugänglich auf GitHub unter der URL *https://github.com/ElmarDott/SamplesAndTrainings*.

Fehlervermeidung

In Organisationen, die kritische Anwendungen entwickeln (wie es beispielsweise die NASA tut), ist eine Anforderung an die erstellten Programme, dass sämtliche Compilerwarnungen für ein Release behoben sein müssen. Dies reduziert zusätzlich die Möglichkeit, dass sich Fehler einschleichen können. Deshalb ist es sehr ratsam, stets sämtliche Compilermeldungen in der Ausgabe aktiviert zu haben und diese zeitnah zu beheben. Moderne Compiler geben zudem in ihren Warnungen auch Hinweise dazu, wie eine Anweisung eindeutiger und somit besser notiert werden kann.

Ein Klassiker in Java ist der Diamond-Operator <>, der Collections (Sammlungen) typisiert. Wird diese Typisierung bei der Deklaration nicht mit angegeben, kann es zu Mehrdeutigkeiten führen, wenn eine Collection mit unterschiedlichen Objekttypen initialisiert wird – ein Umstand, den es verständlicherweise zu vermeiden gilt.

Viele der heutzutage etablierten Praktiken und Methoden haben ihre Wurzeln im *Extreme Programming* (XP). Wegen der starken Ausrichtung auf eine hohe Qualität hat bei diesem Paradigma das Testen einen sehr hohen Stellenwert. Kurze Iterationen, Continuous Integration und Test-Driven Development haben ihre Ursprünge in XP. Viele Praktiken finden sich in abgewandelter Form auch in heute weitverbreiteten agilen Vorgehensmodellen wie Scrum wieder. Dies ist nicht sehr verwunderlich, denn XP und dessen Grundsätze sind im agilen Manifest stark verankert. XP zählt übrigens zu den ersten agilen Methoden.

4.1 Komponententests und das Test-Driven Development

Ein Standardwerk zu testgetriebener Entwicklung erschien bereits im Jahre 2002. Es stammt von Kent Beck und trägt den Titel »Test-Driven Development by Example«. Es besagt, kurzgefasst, dass Sie zuerst die Testfälle schreiben sollten, um anschließend die Funktionalität so zu implementieren, dass sämtliche Tests bestanden werden.

Der größte Kritikpunkt an der vorgeschlagenen Methode ist der enorme Aufwand, mit der das Paradigma umgesetzt werden sollte. Getreu dem Motto »Nichts ist so sicher wie die Änderung« ist die umfangreiche Planung der Testfälle vor der Implementierungsphase für viele Praktiker kaum umsetzbar.

Dennoch ist die Notwendigkeit automatisierter Softwaretests unumstritten. Unternehmen stellen vermehrt zusätzliches Budget für das Testen in ihren Projekten bereit. Aus eigener Erfahrung bin ich allerdings zu der Erkenntnis gekommen, dass man als Developer weder die Erlaubnis des Vorgesetzten benötigt noch eine große Planung durchführen muss, um testgetrieben zu entwickeln.

Die Entwicklungswerkzeuge, die heutzutage zur Verfügung stehen, gestatten es Ihnen, in weniger als fünf Minuten den ersten erfolgreichen Unit-Test zu schreiben.

4.1.1 Erste Schritte zu einer Automatisierung

Das übliche Vorgehen, wenn man eine Funktion programmiert, besteht darin, zuerst die Logik zu schreiben, um diese dann in eine Benutzeroberfläche einzubinden. Anschließend führt man den Code aus und überprüft das Ergebnis. Kommt es zu einem unvorhersehbaren Verhalten, wird an wichtigen Stellen im Quelltext eine Konsolenausgabe der aktuellen Variablenbelegung eingestreut, um die verschiedenen Zwischenschritte zu beobachten. Erfahrene Entwickler und Entwicklerinnen nutzen zu diesem Zweck auch Debugger, um Klarheit über die Arbeitsweise der eigenen Implementierung zu erhalten.

Alles in allem ist das bereits ein testgetriebener Ansatz, wenn auch ein sehr umständlicher. Das größte Problem ist, dass die betroffenen Codebereiche nach Fertigstellung der Funktionalität meist nicht mehr beachtet werden. Ein automatisierter Unit-Test wird hingegen stetig ausgeführt und gibt sofort Rückmeldung darüber, ob eine Änderung möglicherweise Seiteneffekte auf bereits abgeschlossene Arbeiten hat. Es handelt sich also auch um eine Investition in die Zukunft.

Sehr oft wurde ich dank meiner Unit-Tests vor unliebsamen Überraschungen bewahrt, da sie mich rechtzeitig auf Schwierigkeiten hingewiesen haben und ich diese beseitigen konnte, bevor sie zu ernsthaften Problemen herangewachsen sind. Eine solche Situation tritt häufiger ein, als man meint. Besonders nach umfangreichen Anpassungen oder Änderungen ist es ein beruhigendes Gefühl, zu sehen, dass nichts zu Bruch gegangen ist.

Verändern Sie Ihr Vorgehen, und binden Sie die Implementierung nicht mehr wie gewohnt in die GUI der Anwendung ein, um deren Verhalten zu inspizieren. Wenn Sie stattdessen einen Unit-Test nutzen, lösen Sie gleich mehrere Probleme auf einmal. Wichtigster Punkt ist, dass mit dem Erstellen des Unit-Tests eine automatisierte Routine formuliert wurde, die beliebig oft ausgeführt werden kann. Sie erreichen damit zwei wichtige Ziele:

- Wiederholbarkeit
- Reproduzierbarkeit

Ein weiterer Effekt, der sehr früh zu beobachten ist, ist die Beschleunigung der eigenen Produktivität. Der Grund besteht vor allem darin, dass das Bereitstellen der Funktionalität innerhalb der GUI nicht immer so trivial ist, wie es im ersten Moment erscheinen mag. Menüeinträge, Berechtigungen und Eingabemasken müssen erzeugt werden. Zudem müssen Sie sicherstellen, dass diese Änderungen nicht aus Versehen in die Produktion geraten. Oft warten Entwickler auch mit dem Beginn einer Aufgabe, bis die zugehörige Benutzerschnittstelle verfügbar ist. Das kann bei kurzen Iterationszyklen und hinreichender Komplexität der Aufgabe das Zünglein an der

Waage sein, weshalb Arbeiten nicht rechtzeitig fertig werden. Sie wurden einfach zu spät begonnen.

Sie erkennen schon sehr deutlich, in wessen Obhut sich Unit-Tests befinden: Ganz klar ist das Mitglied des Entwicklungsteams, das eine Anforderung der Fachabteilung umsetzt, komplett für die zugehörigen Testfälle verantwortlich. Gibt es spätere Änderungen, Ergänzungen oder Erweiterungen einer bereits umgesetzten Anforderung, die von anderen Mitgliedern des Teams umgesetzt werden, so liegt es in deren Verantwortung, die vorhandenen Tests entsprechend anzupassen.

Kurzanleitung zum Schreiben testbarer Methoden

Die Art und Weise, wie eine Funktion implementiert wurde, beeinflusst maßgeblich ihre Testbarkeit. Wie Sie bereits wissen, basieren Tests darauf, dass die Ausgabe einer Funktion mit einem Erwartungswert verglichen wird. Hat die Funktion aber keinen Rückgabewert und ist somit `void`, ist es sehr schwer, einen Vergleich zu artikulieren. Ähnliche Schwierigkeiten bereiten Methoden, deren Sichtbarkeit auf `private` festgesetzt ist, da diese per Definition außerhalb einer Klasse nicht aufgerufen werden können.

Als Faustregel gilt, dass eine Methode stets einen Eintrittspunkt und einen Austrittspunkt haben sollte. Das `return`-Statement kommt einem zu Recht als Erstes in den Sinn, wenn es um Programmaustrittspunkte geht. Aber auch Exceptions bedürfen besonderer Aufmerksamkeit, da sie den Programmfluss verändern können. Als guter Stil hat es sich bewährt, in der ersten Zeile des Methodenrumpfs die Variable des Rückgabetyps mit einer `default`-Belegung zu initialisieren. Diese Variable wird dann über den Programmfluss manipuliert. Das folgende Codebeispiel verdeutlicht den Sachverhalt und kann auf die meisten Programmiersprachen angewendet werden:

```
public boolean foo(Boolean condition) {
    boolean success = false;
    try {
        if(condition) {
            success = true;
        }
    } catch(Exception ex) {
        System.err.println(ex.getMessage());
     }
    return success;
}
```

Auch wenn die Entwickler die Tests für ihre Implementierung selbst zu verantworten hat, bedeutet es nicht, dass diese ungeprüft akzeptiert werden können. Im Rahmen einer Code-Review müssen auch die zugehörigen Tests beachtet werden, denn auch

in einen Test können sich Fehler einschleichen, wodurch das Ergebnis falsch und der Test unzuverlässig ist.[1]

4.1.2 Testgetriebene Entwicklung mit JUnit

Bisher habe ich bewusst auf konkrete Codebeispiele verzichtet. Ganz gleich, für welches Test-Framework beziehungsweise für welche Technologie Sie sich entscheiden, die bisherigen Aussagen treffen auf alle zu.

Nun ist es allerdings an der Zeit, von der schnöden Theorie hin zu konkreten praktischen Beispielen zu wechseln. Als Test-Framework dient JUnit (*https://junit.org/junit5*) in der Version 5, das in ein Maven/Java-Projekt eingebunden ist. Die zugehörigen Quelltexte können Sie aus dem eingangs erwähnten GitHub-Repository *SampelsAndTrainigs* herunterladen. Die Beispiele sind nach Kapiteln geordnet.

Um JUnit in ein Maven-Projekt einzubinden, müssen Sie den Inhalt von Listing 4.1 in die *pom.xml* Ihres Projekts eintragen:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
```

1 So ist es mir einmal passiert, dass ich im Eifer des Gefechts eine fehlerhafte Überprüfung für die equal-Funktion zweier Objekte implementiert habe. Bei dem Vergleich habe ich das Verhalten Call-by-Reference nicht beachtet und daher ein Objekt falsch kopiert, um eine Identität zu erzeugen. Sinngemäß hatte ich Folgendes formuliert: »Der rote Stift in meiner Hand ist gleich dem roten Stift in meiner Hand.« Das heißt, ich hatte ein Objekt mit sich selbst verglichen. Der Test war also immer erfolgreich. Das, was in einer umgangssprachlichen Formulierung schnell zu sehen ist, hebt sich im Programmcode nicht immer so deutlich ab. Der Fehler wurde dank einer manuellen Review entdeckt und konnte zügig behoben werden.


```
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

Listing 4.1 Integration von JUnit in ein Maven-Projekt (chapter04.1/pom.xml)

Seit Version 5 ist JUnit in mehrere Artefakte zerlegt worden. Die *junit-jupiter-engine* stellt das Test-Framework in der Version 5 dar. Ältere Versionen sind unter der Bezeichnung *vintage* verfügbar. Das Artefakt `api` ist optional, und ich habe es der Vollständigkeit in das Listing mit aufgenommen.

Die Abhängigkeit `params` ist ebenfalls optional. Wird diese eingebunden, können Sie parametrisierte Tests schreiben. Das Artefakt `runner` stellt die Laufzeitumgebung der Test-Engine bereit und ermöglicht die Integration weiterer Test-Frameworks.

Die Maven-Property `${junit.version}` müssen Sie durch eine aktuelle Versionsnummer ersetzen. Beachten Sie außerdem, dass das Framework und der Runner unterschiedliche Versionsnummern haben.

Der Scope *test* bindet die Bibliotheken an die Testausführung und verhindert eine Auslieferung in das fertige Artefakt. Die Testfälle sind im Projektverzeichnis *src/test/java* zu finden. Mehr Schritte sind zur Vorbereitung nicht notwendig, und JUnit kann sofort verwendet werden.

Parametrisierte Tests mit JUnit 5

Die in JUnit zur Verfügung stehenden Testparameter erlauben es, Testdaten zu definieren, die wiederum im Test verwendet werden können. Was es damit auf sich hat, demonstriert ein kleines Beispiel, das alle Funktionsweisen für JUnit 5 kompakt demonstriert. Das Originalbeispiel, das noch auf JUnit 4 basierte, und weitere Informationen finden Sie auf DZone unter *https://dzone.com/articles/junit-parameterized-test*.

```
package org.europa.together.chapter04;

import java.util.stream.Stream;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;
```

```
@RunWith(JUnitPlatform.class)
public class ParameterizedJUnitTest {

    static Stream<Arguments> arrayStream() {
        return Stream.of(
                Arguments.of(2, 3, 6), // {0} = a
                Arguments.of(5, 3, 15), // {1} = b
                Arguments.of(6, 6, 36) // {2} = y
        );
    }

      @ParameterizedTest(name = "{index}: Test with {0} x {1} = {2} ")
    @MethodSource("arrayStream")
    void multiplyTest(int a, int b, int y) {
        ToTestClass tester = new ToTestClass();
        assertEquals(y, tester.multiply(a, b));

    }

    // class to be tested
    private class ToTestClass {

        public int multiply(int i, int j) {
            return i * j;
        }
    }
}
```

Listing 4.2 Testklasse mit einem parametrisierten Test

Um einen parametrisierten Test schreiben zu können, müssen Sie der Testklasse die Annotation `@RunWith` hinzufügen. Als nächstes gilt es in der Methode `arrayStream()` die Testdaten zu erzeugen. Der eigentliche Test `multiplyTest` wird durch die Annotation `@ParameterizedTest` anstatt `@Test` kennzeichnet. Die nachfolgende Annotation `@MethodSource` legt fest, dass die Parameter der Tests aus der Methode `arrayStream()` zu beziehen sind. Damit alles in eine Datei passt, ist das SUT `ToTestClass` mit der zu testenden Methode `multiply()` als innere Klasse definiert.

Die Zuordnung der einzelnen Parameter geschieht über die Methodensignatur und deren drei Integer-Variablen *a*, *b* und *y*. Die Reihenfolge der Parameter korrespondiert mit der Reihenfolge der im Stream erzeugten Argumente. Ich habe Ihne die Verknüpfung im Listing über Kommentare deutlich gemacht. Als Ausgabe erhalten Sie:


```
1: Test with 2 x 3 = 6
2: Test with 5 x 3 = 15
3: Test with 6 x 6 = 36
```

Abbildung 4.2 zeigt, wie in der Entwicklungsumgebung NetBeans die Ausgabe einer Testklasse dargestellt wird. Diese Ansicht erreichen Sie mittels Rechtsklick auf das Editorfenster mit dem Quelltext der Testklasse. Im Pop-up-Menü müssen Sie dann den Eintrag TestFile auswählen. Dann werden die Tests dieser einzelnen Klasse ausgeführt. Sämtliche vorhandenen Testmethoden sind im Fenster Test Results aufgelistet und zeigen an, ob der Test bestanden wurde.

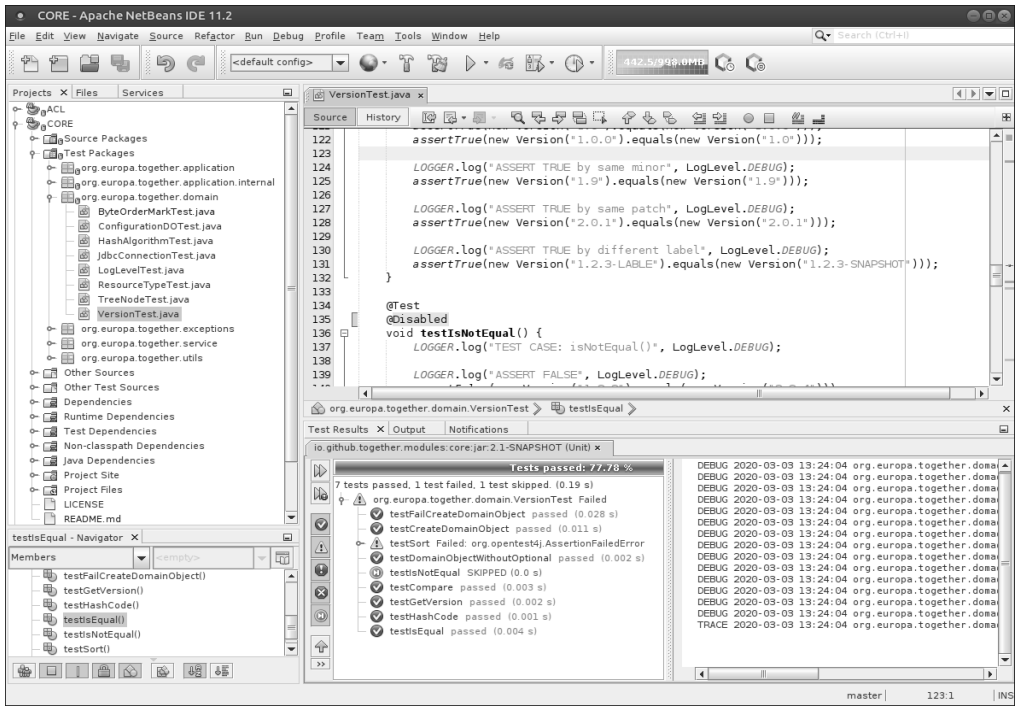

Abbildung 4.2 Testgetriebene Entwicklung in der NetBeans-IDE

Die Konsolenausgabe des fehlgeschlagenen Tests zeigt der rechte Bereich der Test Results an. Für das gewählte Beispiel ist der Test der Methode `sort()` fehlgeschlagen und verweist auf die Zeile 199 in der Testklasse, die einer genaueren Inspektion bedarf. Aber auch über die Annotation `@Disabled` deaktivierte Tests sind in dieser Darstellung hervorgehoben, wie es für die Methode `testIsNotEqual()` der Fall ist. Sie sehen also, wie nahtlos sich das TDD-Paradigma in den Arbeitsfluss integrieren lässt. Dabei sind Sie auch nicht ausschließlich auf NetBeans angewiesen. Sowohl IDEA IntelliJ als auch Eclipse haben ähnliche Funktionen.

Mit dem erworbenen Wissen wollen wir uns nun dem Schreiben von aussagekräftigen Testfällen widmen. Wichtigster Grundsatz ist die strikte Anwendung des KISS-Prinzips: *Keep it Simple, Stupid*.

Bedenken Sie einmal folgende Situation: Für die Vergleichsfunktion `equals()` schreiben Sie eine einzige Testmethode, die alle möglichen Optionen überprüft. Schlägt nun eine einzige Bedingung in dieser Testmethode fehl, werden die nachfolgenden Bedingungen nicht ausgeführt. Daraus ergeben sich nun mehrere Schwierigkeiten, die es zu beherrschen gilt: Eine sehr lange und komplexe Testmethode muss im Fehlerfall mühselig interpretiert werden, um die Ursache für den Fehlschlag zu ergründen. Je einfacher Sie also eine Testmethode formuliert haben, umso leichter lässt sich herausfinden, was das mögliche Problem ist.

Das hat einen merklichen Einfluss auf die Produktivität. Kurze Testmethoden lassen sich schneller formulieren, ihre Intention ist leichter durchschaubar und ihre Ergebnisse lassen sich einfacher interpretieren. Das vereinfacht auch die Wartung und das Hinzufügen neuer Tests.

Ein anderer, weniger bedachter Punkt sind mögliche indirekte Abhängigkeiten. Dieses Problem tritt auf, wenn eine Bedingung fehlschlägt, Sie die Ursache beheben, aber die Lösung zu einem neuen Fehler führt, den Sie so nicht bedacht haben, weil er nicht explizit angezeigt wurde. All dies lässt sich vermeiden, wenn Sie zu einer Anforderung mehrere Testfälle bereitstellen.

Bei der Benennung der Tests müssen Sie auf eine aussagekräftige Namensgebung achten, wie Sie es aus Ihren Coding-Style-Guides bereits gewohnt sein sollten. Das verbessert die Lesbarkeit signifikant.

Bezogen auf das Beispiel des Testens einer Vergleichsmethode, sind mindestens zwei separate Testfälle notwendig:

- Ein Testfall deckt den sogenannten *Happy Path* ab und definiert alle Varianten, für die der Vergleich den Wahrheitswert *true* annimmt.
- Der zweite Testfall beschreibt im *False Positive* das Verhalten im Fehlerfall. False Positive bedeutet, dass der Fehler erwartet, ja sogar bewusst herbeigeführt wurde. So kann festgestellt werden, ob die vorgesehene Fehlerbehandlung wie erwünscht funktioniert.

Wie viele Testfälle werden für eine Methode benötigt?

Um die Anzahl der benötigten Testfälle grob abschätzen zu können, ist die ermittelte zyklomatische Komplexität nach Thomas McCabe eine hilfreiche Orientierung. Die Komplexität einer Funktion lässt sich bestimmen, indem man sämtliche Verzweigungen und Iterationen zählt und zu diesem Wert 1 addiert. Die so erhaltene Zahl korrespondiert mit den benötigten Testfällen, da jede Alternative in der Programmausführung berücksichtigt werden muss.

Das folgende Beispiel hat eine Komplexität von 2:

```
int foo(Boolean condition) {
    int count = 1;
    if(!condition) {
        count++;
    } else {
        count++:
    }
    return count;
}
```

Für aussagekräftige Tests müssen sämtliche Zeilen der Methode durchlaufen werden. Um eine Testabdeckung von 100 % zu erreichen, müssen Tests formuliert werden, mit denen alle Variationen durchprobiert werden, die den `if`-Zweig erreichen. In einem weiteren Test müssen Sie die Bedingung so wählen, dass der `else`-Zweig ebenfalls ausgeführt wird.

Das Thema *Testcoverrage* greifen wir in Kapitel 10, »Qualitätskontrolle«, erneut auf und beschäftigen uns dann mit Qualitätsbewertungen von Programmcode. Merken Sie sich aber schon einmal die *TDD-Regel*: Wird der Test spezifischer und somit detaillierter, ist die zugehörige Implementierung generalisierter und abstrakter.

Ich hatte bereits angedeutet, wie schwierig es sich gestalten kann, wenn Sie Methoden testen müssen, die als `private` definiert wurden. Auch haben wir bereits eine Vorstellung davon gewonnen, dass der Rückgabetyp `void` ebenfalls problembehaftet ist. Die einzige Möglichkeit, dem beizukommen, besteht darin, die Implementierung entsprechend so zu gestalten, dass Codebereiche mit diesen Eigenschaften erreicht werden können. Listing 4.3 definiert eine Klasse zur Potenzierung:

```
public class SystemUnderTest {

    private int parameter;

    public int power() {
        return calculate(parameter, parameter);
    }
    public void setParameter(int parameter) {
        this.parameter = parameter;
    }
    public int getParameter() {
        return this.parameter;
    }
```

```
    private int calculate(int operand01, int operand02) {
        return operand01 * operand02;
    }
}
```

Listing 4.3 »chapter04.1/SystemUnderTest.java«

Um die gegebene Klasse umfangreich testen zu können, sollten Sie schrittweise vorgehen. Der Setter für die Variable hat den Rückgabewert `void`. Allerdings hat der Aufruf eine Veränderung des gesamten Objekts zur Folge, da es die Klassenvariable `parameter` mit einem Wert belegt. Dieser Wert lässt sich über den Getter wiederum auslesen – womit wir unseren ersten Testfall formulieren können:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class SystemUnderTestTest {

    @Test
    public void testGetterAndSetter() {
        // Arrange
        int parameter = 1;
        SystemUnderTest sut = new SystemUnderTest();
        // Act
        sut.setParameter(parameter);
        // Assume
        assertEquals(parameter, sut.getParameter());
    }
}
```

Listing 4.4 »chapter04.1/SystemUnderTestTest.java«, Version 1

Die private Methode `calculate()` ist dazu gedacht, Code-Dubletten zu vermeiden. Sie wird implizit mit den Funktionen getestet, die `calculate()` aufrufen. In unserem Beispiel ist dies die Methode `power`, die durch einen eigenen Testfall überprüft wird. Ein weiterer Testfall soll uns einen Eindruck davon vermitteln, was passiert, wenn die Methode `power` nicht korrekt über die Methode `setParameter()` initialisiert wurde. Entdeckt man in einer Klasse Methoden und Attribute, die als `private` markiert wurden, aber nirgendwo anders im Quelltext Verwendung finden, so hat man höchstwahrscheinlich sogenannten toten Code identifiziert und sollte diesen aus der Klasse entfernen. Dazu setzt man vorsichtshalber einen separaten Commit ins Repository, um die Änderung zu dokumentieren und ein Rollback zu ermöglichen, falls die Löschung ein Versehen war:


```
@Test
void testCalculatePower() {
   // Arrange
   int result = 16;
   SystemUnderTest sut = new SystemUnderTest();
   // Act
   sut.setParameter(4);
   // Assume
   assertEquals(result, sut.power());
}

@Test
void testFailCalculatePower() {
    // Arrange
    SystemUnderTest sut_ = new SystemUnderTest();
    // Act
    int result = sut_.power();
    // Assume
    assertEquals(0, result);
}
```

Listing 4.5 »chapter04.1/SystemUnderTestTest.java«

Sicher haben einige von Ihnen für die Variante ohne eine Initialisierung von `parameters` über seine Setter als Rückgabe `null` erwartet. Da es sich bei `int` aber um einen primitiven Datentyp handelt, lautet das Ergebnis 0. Das Autoboxing in Java für den primitiven Datentyp und die Klasse `Integer` bewirkt das gleiche Ergebnis. Das vollständig konfigurierte und lauffähige Projekt finden Sie im Repository im Unterverzeichnis *chapter04.1*.

Der Hamcrest-Stolperstein

In der Praxis ist es eher mühselig, für Standardmethoden immer wieder gleiche Testroutinen zu schreiben. Um Default-Implementierungen wie Getter, Setter, `hash`, `equals`, `toString` und Konstruktoren mit nur wenigen Codezeilen zu testen, bedienen Sie sich der Bibliothek *bean-matchers*, die Sie hier finden: *https://github.com/orien/bean-matchers*

```
<dependency>
    <groupId>com.google.code.bean-matchers</groupId>
    <artifactId>bean-matchers</artifactId>
    <version>${beanmatcher.version}</version>
    <scope>test</scope>
</dependency>
```

Die Implementierung dieser Bibliothek basiert wie viele andere Test-Frameworks auch auf *Hamcrest*. Manchmal kommt es aber vor, dass Tests aus der `bean-matchers`-Bibliothek fehlschlagen und die Fehlermeldung sehr unverständlich ist.

Die Lösung dieses Problems ist sehr einfach. Da Maven die Reihenfolge der in der POM notierten Abhängigkeiten entsprechend der Reihenfolge der Notation auflöst, sollte die `bean-matchers`-Library möglichst zu Beginn in der Liste der Test-Frameworks eingetragen werden. Die Artefakte für JUnit wiederum sollten möglichst das Ende der Eintragungen mit dem Scope `test` bilden.

Ich habe bereits erwähnt, dass Komponententests sehr rudimentär sind, auf Methodenebene operieren und daher in ihrer Ausführung recht schnell sind. Die Ausführungsgeschwindigkeit aller Testfälle ist ein wichtiges Detail für die tägliche Arbeit eines Entwicklerteams.

Wenn die Applikation als Monolith konzipiert ist und zudem auch eine große Codebasis von mehreren Hunderttausend Codezeilen aufweist, hat sich naturgemäß auch ein großer Umfang an Testfällen angesammelt. Das kann dazu führen, dass das Kompilieren auf dem lokalen Entwicklungsrechner schnell die 5-Minuten-Grenze überschreitet – ein Umstand, der den Arbeitsfluss beim Programmieren stark beeinträchtigt.

Um eine möglichst geringe Ausführungszeit zu erzielen, sind kostenintensive Operationen wie Datenbankabfragen oder Netzwerkabfragen in Unit-Tests sehr verpönt. Eine Ansicht, die ich nur halbwegs teile, denn gerade Datenbankzugriffe sind wichtige Operationen in Programmen. Besonders die korrekte Selektion definierter Datensätze aus der Persistenzschicht halte ich für essenzielle Operationen, die zwar kostenintensiv sind, aber unverzichtbar.

Sie sollten daher den Funktionsumfang eines Artefakts so eingrenzen, dass die Testausführungen eine Dauer von fünf Minuten nicht überschreiten. Sollte sich dies als unmöglich erweisen, lassen sich die Unannehmlichkeiten nur durch ein umfangreiches Refactoring beheben.

Neben dem Eliminieren toter beziehungsweise ungenutzter Fragmente gilt es ebenso, Möglichkeiten zu finden, wie sich der Monolith in kleinere handhabbarere Einheiten zerlegen lässt. Damit ist aber keine Migration auf eine modulare Architektur gemeint, die wiederverwendbare Artefakte zur Folge hat. Kleinere unabhängige Einheiten, bezogen auf die fachliche Zuordnung und Änderungsfrequenz, sind zwar im weitesten Sinn auch Module, aber ihr Zweck ist die Reduzierung des Umfangs, um einen besseren Überblick zu gewährleisten und die Verantwortlichkeiten deutlicher zu trennen. Trotz einer solchen Aufteilung bleibt die Anwendung in ihren Merkmalen weiterhin ein Monolith.

Ein anderer Aspekt ist das Verhalten der Build-Logik, wenn Testfälle fehlschlagen. Viele Werkzeuge haben eine Einstellung, die es gestattet, den Build-Prozess auch bei einem Fehlschlag von Testfällen fortzuführen. Auf diese Weise erhalten Sie einen vollständigen Bericht über alle vorhandenen Fehler. Ein Abbruch der Artefakt-Erstellung sollte nur dann in Erwägung gezogen werden, wenn ein Release erzeugt wird. Komponententests, die aufgrund fehlender Implementierung nicht bestanden werden können, sollten Sie deaktivieren. Dies stellt sicher, dass diese weiterhin Beachtung finden, und signalisiert auch, dass noch Arbeiten dazu notwendig sind.

Eine andere Taktik ist, nicht verfügbare Funktionalität durch Mock-Objekte zu simulieren. Diesen Ansatz würde ich weniger empfehlen, da das Risiko besteht, dass der Entwickler im Eifer des Gefechts vergisst, den entsprechenden Test-Case anzupassen. Gerade dann, wenn Fertigstellungstermine vor der Tür stehen, ist in vielen Projekten ein sehr hoher Stresslevel präsent, der schnell zu Fehlern führen kann.

```
--- maven-surefire-plugin:3.0.0-M4:test (default-test) @ tdd-testing ---

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.europa.together.chapter03.SystemUnderTestTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s - in org.europa.together.chapter03.SystemUnderTest

Results:

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0


--- maven-jar-plugin:2.4:jar (default-jar) @ tdd-testing ---
Building jar: /media/storage/BACKUP/Publications/Buchprojekt - CI mit Jenkins/CodeSamples/chapter03.1/target/tdd-testing-1

--- maven-install-plugin:2.4:install (default-install) @ tdd-testing ---
Installing /media/storage/BACKUP/Publications/Buchprojekt - CI mit Jenkins/CodeSamples/chapter03.1/target/tdd-testing-1.0-
Installing /media/storage/BACKUP/Publications/Buchprojekt - CI mit Jenkins/CodeSamples/chapter03.1/pom.xml to /media/verac
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.927 s
Finished at: 2020-03-05T15:19:07-06:00
```

Abbildung 4.3 Konsolenausgabe des »maven-surefire-plugin« für Unit-Tests

Wie Sie in Abbildung 4.3 sehen können, ist die Konsolenausgabe aller Ergebnisse eines Artefakts zwar nützlich, aber wenn es darum geht, einen Gesamtüberblick zu erhalten, kann es schnell etwas unübersichtlich werden. Hier empfiehlt es sich, im Maven-Site-Life-Cycle eine für Menschen lesbare Übersicht generieren zu lassen. Dies erreichen Sie mit nur wenigen Zeilen in der POM:

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>${surefire.version}</version>
```

```
        </plugin>
    </plugins>
</reporting>
```

Listing 4.6 Maven-Report-Generierung

Mit dieser Einführung zum Thema Unit-Tests können Sie bereits Testfälle für Ihre Funktionen schreiben. Damit haben Sie die einfachste Möglichkeit kennengelernt, um automatisiert festzustellen, wann sich das Verhalten Ihrer Anwendung verändert hat.

4.2 Akzeptanztests und Behavior-Driven Development

Abnahmetests oder auch Akzeptanztests beschreiben eine Kategorie von Tests, die festlegen, unter welchen Bedingungen ein Auftraggeber eine Software als fertiggestellt anerkennt. Die Formulierung der Testfälle liegt deshalb bei der Fachabteilung, die auch die Anforderungen an das zu erstellende Programm festlegt.

Klassisch gehört diese Methode zum Black-Box-Testen, da der Auftraggeber sich üblicherweise nicht dafür interessiert, wie die Implementierung beispielsweise eines Newsletters realisiert wurde. Ihm geht es einzig darum, welche Interaktionen mit dem Programm notwendig sind, damit ein Newsletter unfallfrei versendet wird, und wie das Programm auf fehlerhafte Eingaben reagiert. Deswegen werden Akzeptanztests oft durch *Use-Cases*, *User-Stories* oder *Szenarien* beschrieben, denen eine Interaktion zugrunde liegt.

In Bezug auf die bereits vorgestellte Test-Pyramide handelt es sich bei Abnahmetests um die Prüfverfahren, die eine hohe Integrationsstufe haben und folglich auch eine lange Ausführungszeit benötigen. Deswegen werden Abnahmetests im Vergleich zu Unit-Tests eher selten durchgeführt. Üblicherweise durchläuft ein fertiggestelltes Release die Abnahmetests, damit man entscheiden kann, ob noch Fehler gefunden werden, die eine Abnahme verhindern.

Wird ein schwerwiegender Fehler entdeckt, der einen produktiven Einsatz des Programms nicht zulässt, wird umgehend eine Korrektur zu diesem Release erstellt, und die Prozedur der Abnahmetests beginnt von Neuem. Das bereits zu Beginn dieses Kapitels besprochene Beispiel mit der falsch berechneten Mehrwertsteuer gehört zur Klasse der schwerwiegenden Fehler.

Einen tieferen Einblick in die Vorgänge, wie eine Anwendung ihren Weg vom Release in die Produktion findet, erhalten Sie im nächsten Kapitel. Dort widmen wir uns den Fragen des Release-Managements.

Behavior-Driven Design (BDD)

BDD oder verhaltensgetriebenes Design ist ein Paradigma, das eine Brücke zwischen Nicht-Technikern, also der Fachabteilung, und den Technikern im Entwicklungsteam schlägt. Erstmals wurde es 2006 von Dan North in einem Artikel im *Better Software*-Magazin vorgestellt. North ist ebenfalls Entwickler des BDD-Frameworks *JBehave*.

Während der Anforderungsanalyse entstehen Beispiele, die als Szenarios auf der Basis von Wenn-dann-Sätzen in Textform festgehalten werden. Später werden diese dann als automatisierter Test umgesetzt.

Das folgende Szenario beschreibt einen erfolgreichen Login, in der Form, wie es oft von Fachabteilungen formuliert wird:

- **GEGEBEN** – Der Nutzer ist registriert
- **UND** – der Account ist aktiv
- **UND** – das Passwort ist korrekt
- **WENN** – Nutzer sich am System anmeldet,
- **DANN** – gestatte Zugriff.

4.2.1 Organisation von Akzeptanztests

Wenden wir uns an dieser Stelle nun den technischen Details zu, und klären wir die Frage, wo man im SCM-Repository Akzeptanztests organisieren sollte.

Die wichtigste Bedingung ist, dass diese Testkategorie nicht gemeinsam mit Unit-Tests während des Standard-Builds ausgeführt wird. Viele Unternehmen organisieren ihre Abnahmetests in eigenständigen Projekten, die durch eine Testabteilung als interne Qualitätskontrolle verwaltet werden. Üblicherweise haben die Anwendungsentwickler keinen Anteil an der Formulierung der Akzeptanztests, weswegen die Tests auch in aller Regel nicht im Code-Repository der Anwendung zu finden sind.

Das SUT wird in solchen Projekten nur als Abhängigkeit eingebunden. Daher werden einzig die Binärartefakte benötigt, um entsprechende Testszenarien formulieren zu können.

Der Vorteil dieser Struktur ist die physische Trennung des Quelltextes von den Testszenarien. Dies vereinfacht die Zusammenarbeit mit externen Dienstleistern. Das gilt auch für interne Abteilungen zur Qualitätssicherung wie dem Test-Center. Der Sourcecode bleibt im Hoheitsgebiet der Entwicklungsabteilung und kann durch Außenstehende nicht verändert werden. So werden Firmengeheimnisse gewahrt und spezielle Algorithmen, Implementierungsdetails und anderes Wissen kann nicht so leicht nach außen dringen.

Eine andere Möglichkeit ist es, die Akzeptanztests im selben Verzeichnis wie die Unit-Tests zu organisieren. Das Unterscheidungsmerkmal zwischen beiden Kategorien ist das Postfix im Dateinamen. So zeichnet beispielsweise der Klassenname `MyClassTest` einen Unit-Test aus, und `MyIntegrationIT` mit der Endung `IT` definiert Integrationstests. Problematisch bei dieser Lösung ist die Vermischung von Akzeptanz- und Integrationstest. Eine klare Abgrenzung beider Teststrategien ist nicht immer eindeutig auszumachen.

Testen mit Maven

Während für Unit-Tests das *maven-surefire-plugin* zuständig ist, führt das *maven-failsave-plugin* Integrationstests aus. Die Gundkonfiguration für das Failsafe-Plug-in lautet wie folgt:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>${failsave.version}</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Um eine permanente Ausführung zu verhindern, kann das Plug-in an ein Profil gebunden werden. Sämtliche Testklassen, die mit `IT` beginnen oder enden, werden durch das *maven-failsafe-plugin* ausgeführt. Auch für Integrationstests können in Maven Reports erzeugt werden. Das dafür zuständige Plug-in *maven-surefire-report-plugin* kennen Sie bereits von den Unit-Tests.

Eine Bibliothek, mit der Sie für die Java-Plattform verhaltensgetriebene Tests (BDD) umsetzen können, ist *JGiven*.

4.2.2 Verhaltensgetriebene Tests mit JGiven

JGiven integriert sich bestens in das JUnit-Framework, aber auch eine Kombination mit TestNG ist ohne Weiteres möglich. Zur Präsentation der Ergebnisse steht ein Generator bereit, der eine leicht zu erfassende Übersicht erzeugt. Diese Berichte können mit der Reporting-Engine von Maven ebenfalls erzeugt werden. Dazu muss das

jgiven-maven-plugin konfiguriert werden. Fügen Sie dazu den Inhalt aus Listing 4.7 in die *pom.xml* Ihres Projekts ein:

```
<plugin>
    <groupId>com.tngtech.jgiven</groupId>
    <artifactId>jgiven-maven-plugin</artifactId>
    <version>${jgiven.version}</version>
    <executions>
        <execution>
            <phase>pre-site</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing 4.7 »chapter04.2/pom.xml«

Eine der großen Stärken von JGiven ist die gute Verständlich- und Lesbarkeit der für Menschen generierten Reports. Auch wenn JGiven vornehmlich für die Zusammenarbeit mit JUnit 4 implementiert wurde und die Integration für JUnit 5 als experimentell gekennzeichnet ist, sind mir bei der Kombination aus JGiven und JUnit 5 keine Probleme diesbezüglich untergekommen. Es gibt also keinen Grund, JGiven für Akzeptanztests nicht zu verwenden.

Die *Szenarios*, wie die Testfälle in JGiven genannt werden, werden in Java notiert.

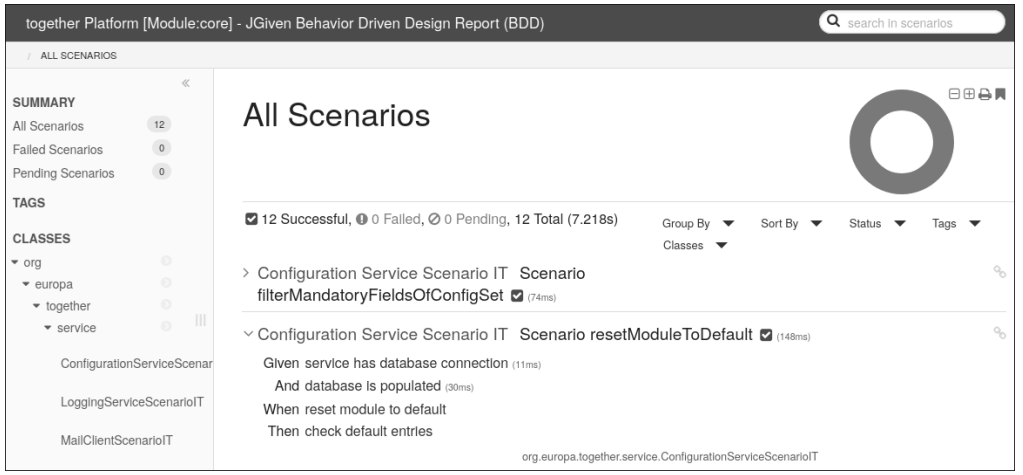

Abbildung 4.4 Testbericht der JGiven-Szenarios

Wie Sie in Abbildung 4.4 an dem Beispielszenario aus meinem GitHub-Projekt *TP-CORE* sehen können, orientiert sich JGiven auch am bereits vorgestellten Grundsatz *Arrange, Act, Assume* (AAA). Lediglich die Benennung ist etwas anders. Dem *Arrange* steht das *Given* gegenüber, *When* ist dem Act zuzuordnen, und *Assume* wird durch *Then* repräsentiert.

Given, *When* und *Then* werden in eigenen Klassen implementiert, die Sie theoretisch auch in anderen Szenarios wiederverwenden können. Eine Mehrfachverwendung ist allerdings mit viel Vorsicht zu genießen und nur dann zu empfehlen, wenn der Tester bereits viel Erfahrung mit dem Schreiben von Tests gesammelt hat. Um erfolgreich wiederverwendbaren Code zu entwerfen, ist es unabdingbar, diesen möglichst einfach zu halten. Schnell kann hier ein komplexes und schwer beherrschbares Konstrukt entstehen.

Wie der Testfall umgesetzt wird, zeigt Ihnen sehr schön das Szenario `resetModuleToDefault`. Dabei sollte es für uns keine Rolle spielen, um was für einen speziellen Use-Case es sich handelt. Vielmehr geht es darum, zu zeigen, wie intuitiv die Szenarienbeschreibung gelesen werden kann und wie leicht mögliche Fehler zu erkennen sind.

Die Bedeutung der Ausgabe lässt sich folgendermaßen lesen:

So gilt das,

- **wenn** die Datenbankverbindung hergestellt wurde
- **und** die korrekten Datensätze vorhanden sind,
- **dann** setze die Einträge eines Moduls zurück auf ihre Default-Werte,
- **folglich** ist die aktuelle Konfiguration überschrieben und in ihre Default-Werte geändert.

4.2.3 Reibungslose Zusammenarbeit dank verständlicher API-Dokumentation

In der kommerziellen Softwareentwicklung sind unterschiedliche Personengruppen in die Erstellung der Akzeptanztests involviert. Die Fachabteilung formuliert die Testanforderung, und der Tester implementiert das Szenario. Ein weiterer Grund, warum die Szenarios nicht durch den Programmierer umgesetzt werden sollten, der die Funktionalität entwickelt hat, ist der Test der Benutzbarkeit der bereitgestellten API.

Das Vorgehen gleicht also auch einem klassischen Vieraugenprinzip. Kann der Tester ein Szenario nicht den Vorgaben entsprechend umsetzen, erhält man einen guten Eindruck von der Vollständigkeit der API und muss an dieser Stelle sehr wahrscheinlich nachbessern. Daher ist eine saubere API-Dokumentation extrem wichtig. Erst wenn ein Fremder aus den verfügbaren Bausteinen das Bild eigenständig zusammensetzen kann, ist das Vorhaben tatsächlich gelungen.

Sehr treffend formulierte einst ein Kollege seinen Missmut über unzureichend formulierte JavaDoc-Kommentare. Er verglich Beschreibungen, die einfach den Methodennamen als Satz formulieren, mit dem Foto eines Hunds, dem ein Zettel mit dem Wort »Hund« auf die Stirn geklebt wurde. Der Mehrwert der Erläuterung im Kommentar ist gleich null.

Bei der Formulierung eines JavaDoc-Kommentars beschreibt bereits der Methodenname, *was* die Funktion ist. Im Kommentar sollten deshalb das *Wie* und das *Warum* festgehalten werden. JavaDoc stellt zur besseren Dokumentation ab Java 8 drei neue Annotationen bereit, die genutzt werden können:

- `@apiNote` enthält Informationen bezüglich der API, zum Beispiel ab welcher API-Version die Methode hinzugefügt wurde.
- `@implSpec` gibt Hinweise zur Implementierung der Spezifikation. Das können Aussagen darüber sein, *wie* sich eine Implementierung verhält.
- `@implNote` kann beispielsweise Informationen enthalten, ob es bei der Verwendung Einschränkungen gibt, die es zu berücksichtigen gilt.

Wie die Java-API die neuen Annotationen verwendet, können Sie sich am Beispiel des Interfaces der ConcurrentMap (*https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html*) anschauen. Das vermittelt Ihnen einen detaillierteren Eindruck davon, wie Sie die vorgestellten Annotationen nutzen können.

Wenn Sie die Funktion `equals` für eine Klasse, die eine Versionsnummer darstellt, mit den Worten »Vergleich der Versionsnummer« beschreiben, dann haben Sie viel erzählt, aber wenig gesagt. Folgende Beschreibung liest sich viel besser: »Versionsnummern mit dem optionalen Zusatz `SNAPSHOT` kennzeichnen Artefakte, die sich im Entwicklungsstadium befinden. Die Ausdrücke 1.0 und 1.0.0 sind äquivalent.«

Von Mocks und Stubs

Mock-Objekte und *Stubs* (dt. »Stummel«) sind sehr hilfreiche Konstrukte, wenn es darum geht, Objekte in einem Test verwenden zu müssen, die sich aber nicht immer ohne Weiteres erzeugen lassen. Dies können Anbindungen an Module sein, die beispielsweise noch gar nicht bereitstehen.

Als Lösung erzeugen Sie ein Objekt, das das Original in seiner Erscheinung simuliert. So praktisch diese Technologie auch ist, sollten Sie ihre Verwendung doch gut abwägen. Wie bei vielen Dingen im Leben lässt die Dosis die Medizin zum Gift werden.

Das bedeutet: Werden zu viele Mocks in Tests verwendet, ist es schwer, eine verlässliche Aussage über das exakte Verhalten zu treffen. Adam Bien verglich in einem Vortrag die Verwendung von Mocks mit einer Testfahrt im Auto, dessen Bremsen »gemockt« wurden: Die Fahrt würde äußerst rasant. Man hat ein Pedal, das aussieht wie eine Bremse und sich auch so anfühlt, aber das Drauftreten bewirkt keine Reaktion.